СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Системный анализ данных в системах поддержки принятия решений представляет собой область знаний, изучающую методы структурирования, обработки и интерпретации информации для повышения качества выбора в условиях неопределённости. В рамках курса рассматриваются основы онтологического моделирования, позволяющего формализовать знания предметной области и обеспечить единое понимание данных различными участниками и системами. Особое внимание уделяется изучению современных эвристических методов оптимизации, которые имитируют процессы, наблюдаемые в природе, и позволяют эффективно решать сложные вычислительные задачи. Метод отжига основан на физической аналогии охлаждения металлов и используется для поиска глобального минимума в пространствах с большим количеством локальных экстремумов. Роевые алгоритмы воспроизводят коллективное поведение децентрализованных систем и направлены на достижение оптимума через взаимодействие большого количества простых агентов. Муравьиный алгоритм имитирует поведение муравьёв при поиске кратчайших путей и демонстрирует способность находить эффективные решения за счёт накопления феромонной информации. Пчелиный алгоритм вдохновлён стратегиями поиска нектара в пчелиных колониях и сочетает разведку новых областей с усилением перспективных направлений. Изучение этих подходов формирует у студента фундаментальное понимание интеллектуальных методов анализа данных и позволяет применять полученные знания для разработки систем поддержки принятия решений. Данные методы рассматриваются на примере нахождения минимума выбранной функции и задачи Коммивояжёра.

1 ОНТОЛОГИЯ

Онтология — это формальное представление знаний о предметной области, включающее объекты, их свойства и отношения между ними.

В условиях цифровизации и интенсивного роста объёмов данных возрастает потребность в формализованном представлении знаний, обеспечивающем их структурирование, интерпретацию и повторное использование в информационных системах. Одним из наиболее эффективных инструментов решения данной задачи являются онтологии, позволяющие описывать предметные области в виде формальных моделей, включающих понятия, их свойства и отношения между ними.

Актуальность разработки и применения онтологий обусловлена необходимостью интеграции разнородных источников данных, повышения семантической совместимости информационных систем и поддержки интеллектуальных методов обработки информации. Онтологический подход широко используется в системах поддержки принятия решений, интеллектуальных поисковых системах, экспертных системах и при построении баз знаний, что делает его востребованным в современных задачах системного анализа.

Объектом исследования в данной выпускной квалификационной работе является организационно-структурная модель медицинского учреждения (поликлинической больницы), представленная в виде онтологии предметной области.

Онтология описывает структуру поликлинической больницы как совокупность взаимосвязанных сущностей, включающих управленческий персонал, медицинских и вспомогательных сотрудников, а также функциональные помещения учреждения. В рамках онтологии формализованы основные классы предметной области: Поликлиническая больница, Сотрудники,

Руководство, Штат и Помещения, а также их иерархические и функциональные отношения.

1.1 Цель и задачи практической работы

Цель работы заключается в формировании умений по применению методов системного анализа данных при проектировании базы знаний для заданной предметной области.

Задачи работы включают:

1. Исследовать предметную область, для которой будет разрабатываться база знаний.
2. Определение состава объектов, их свойств и взаимосвязей.
3. Построение концептуальной модели базы знаний.
4. Формирование связей различных видов между объектами.
5. Перенос базы знаний в инструмент Protégé.
6. Перенос базы знаний в кодовый формат.

1.2 Постановка задачи

В рамках практической работы необходимо выбрать предметную область и выполнить её формализацию в виде базы знаний. Для этого требуется определить множество объектов и их атрибутов, выделить связи между объектами. На основе полученных данных следует построить концептуальную модель базы знаний и спроектировать её структуру, обеспечивающую возможность дальнейшего применения для решения задач системного анализа.

Построить данную модель в инструменте Protege и программном виде на языке Python.

1.3 Проектирование базы знаний

В рамках практической части работы была выбрана предметная область — психиатрическая больница. Для данной области выполнено построение базы знаний, отражающей структуру управления, штат сотрудников и используемые помещения.

Основными классами базы знаний являются:

- «Руководство»;
- «Сотрудники»;
- «Помещения».

Каждый объект обладает набором характеристик, включая ФИО, подчинённых и подчинение, что позволяет формализовать как вертикальные связи управления, так и горизонтальные взаимосвязи между элементами системы.

Построенная иерархическая схема отражает организационную структуру психиатрической больницы: от руководства к сотрудникам и помещениям (Рис.1). Предметом исследования в данной выпускной квалификационной работе являются методы и способы формализации, структурирования и оптимизации онтологической модели организационно-функциональной структуры психиатрической больницы.

В рамках исследования предметом рассмотрения являются отношения между концептами онтологии, их иерархическая организация, а также атрибуты и ограничения, определяющие подчинённость, функциональные роли сотрудников и распределение помещений по назначению. Особое внимание уделяется процессам оптимизации структуры онтологии с целью повышения её согласованности, полноты и пригодности для использования в интеллектуальных информационных системах.

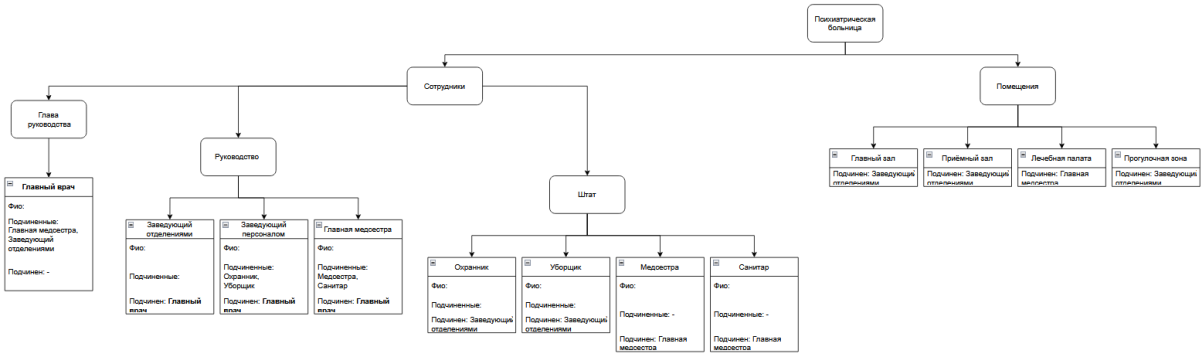


Рисунок 1.1 – Схема базы знаний

1.4 Разработка базы знаний в инструменте Protege

Перенесем построенную базу знаний в инструмент Protege. Для начала построим иерархию классов, отображающую общую структуру.

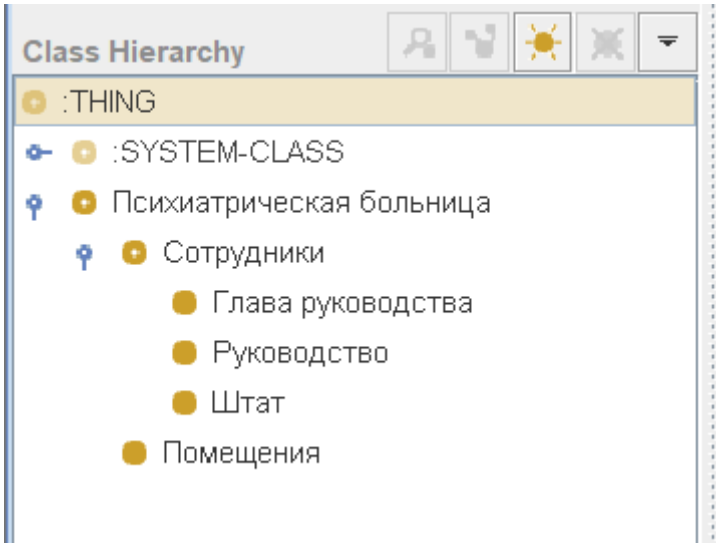


Рисунок 1.2 – Иерархия базы знаний

Также напишем поля, которые будут присвоены к определенным класса, дабы потом реализовать объекты этих классов.

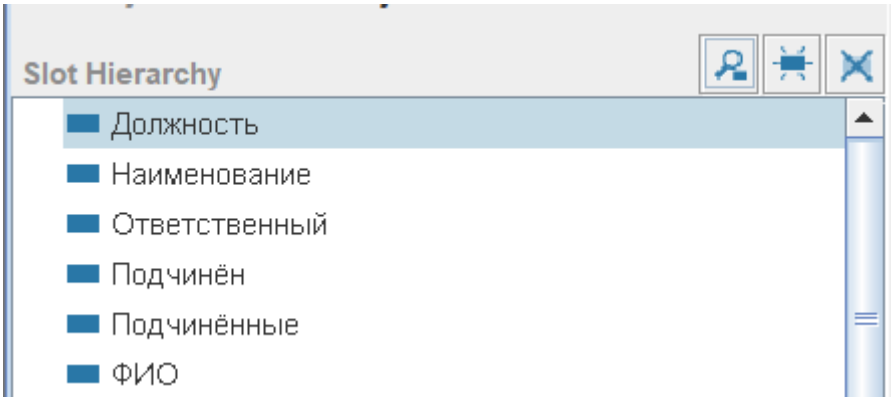


Рисунок 1.3 – Поля используемые в базе знаний

Создадим объекты всех классов данной базы знаний и заполним поля, дабы каждый объект был персонализирован.

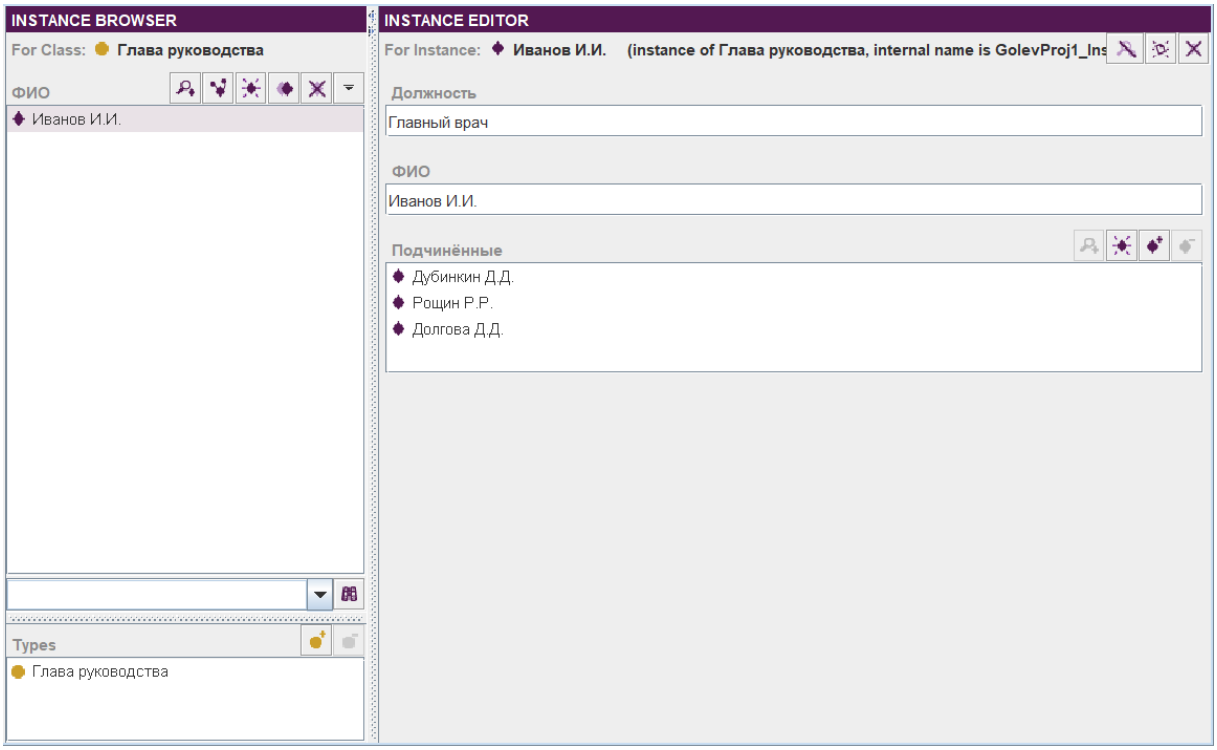


Рисунок 1.4 – Объект класса «Глава руководства »

Далее создадим объект класса Руководство.

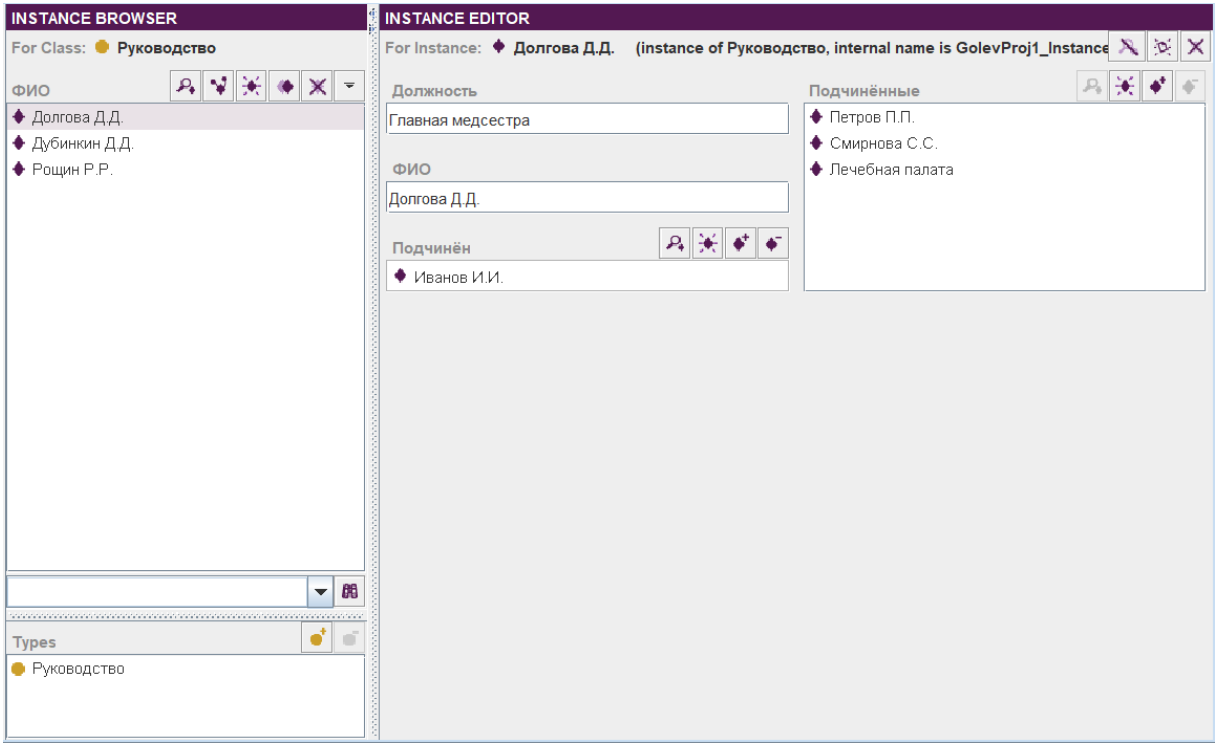


Рисунок 1.5 – Объекты класса “Руководство »

Далее создадим объект класса Штат.

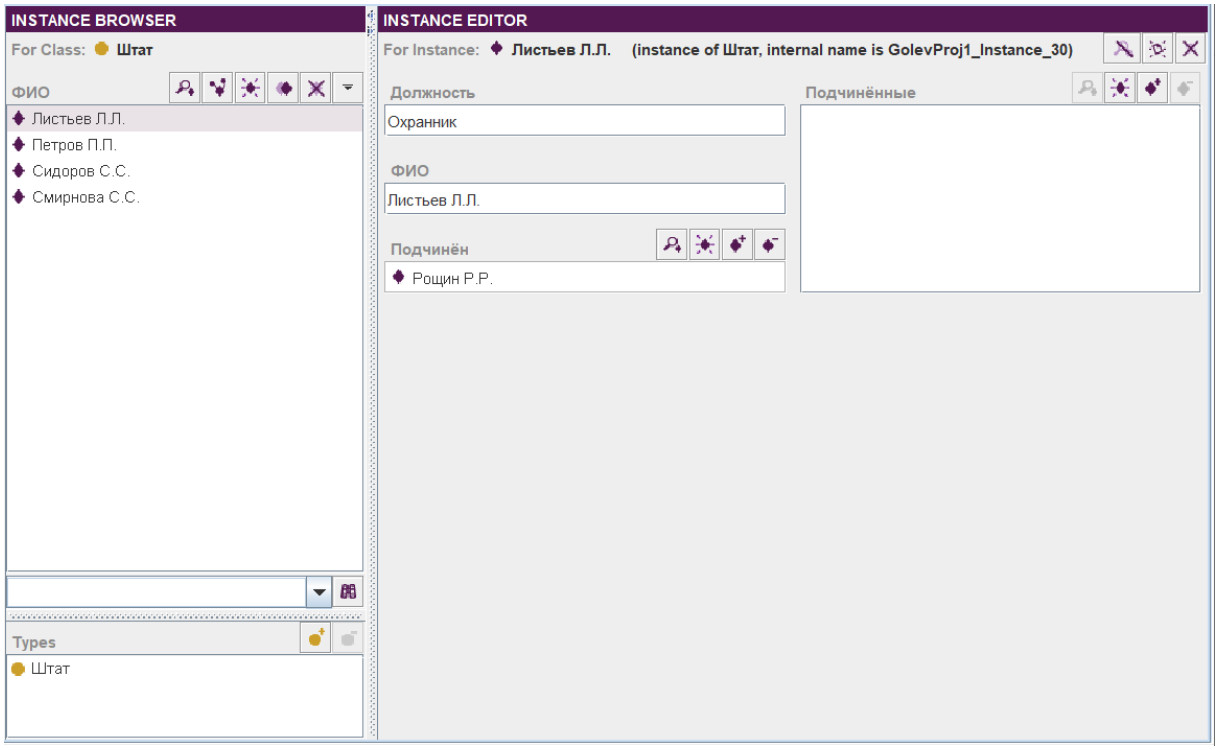


Рисунок 1.6 – Объекты класса «Штат »

Далее создадим объект класса Помещения.

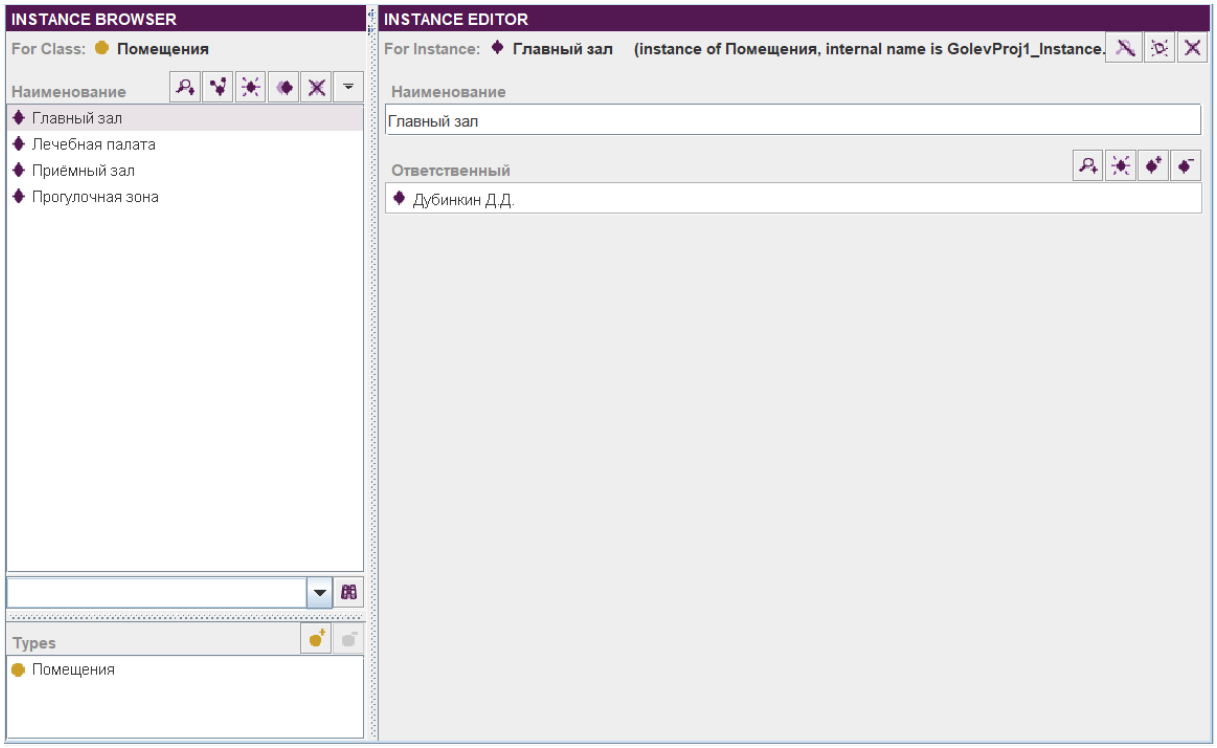


Рисунок 1.7 – Объекты класса «Помещения »

Напишем запросы, с помощью которых можно будет посмотреть связи между объектами различных классов, реализуем связи:

1. Один к одному.
2. Один ко многим.
3. Один к одному через рукопожатие.
4. Один ко многим через рукопожатие.

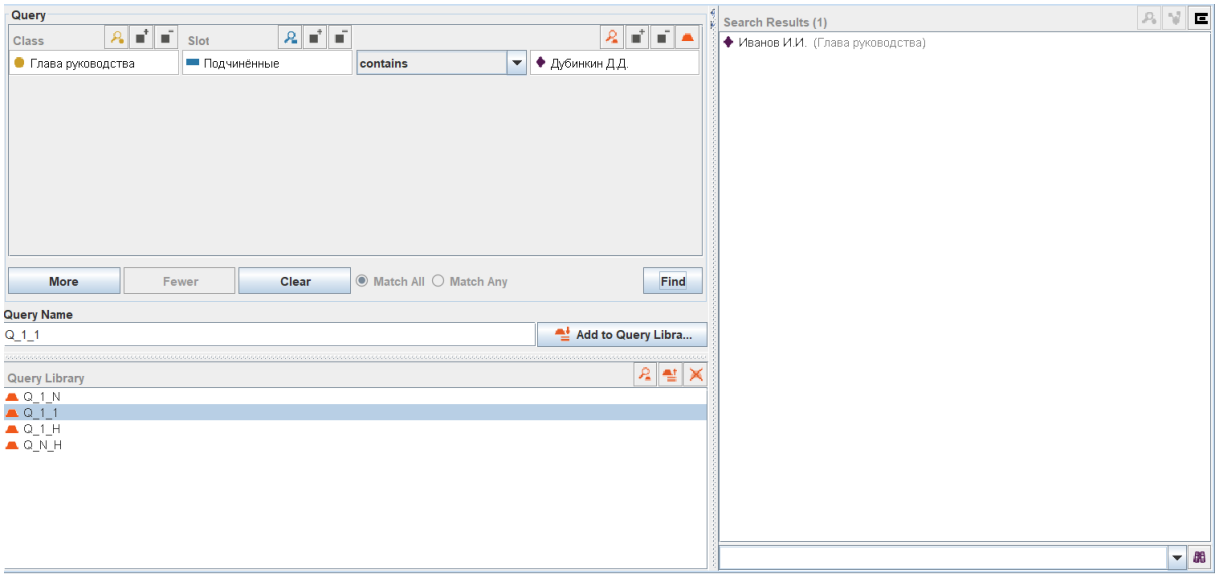


Рисунок 1.8 – Запрос один к одному

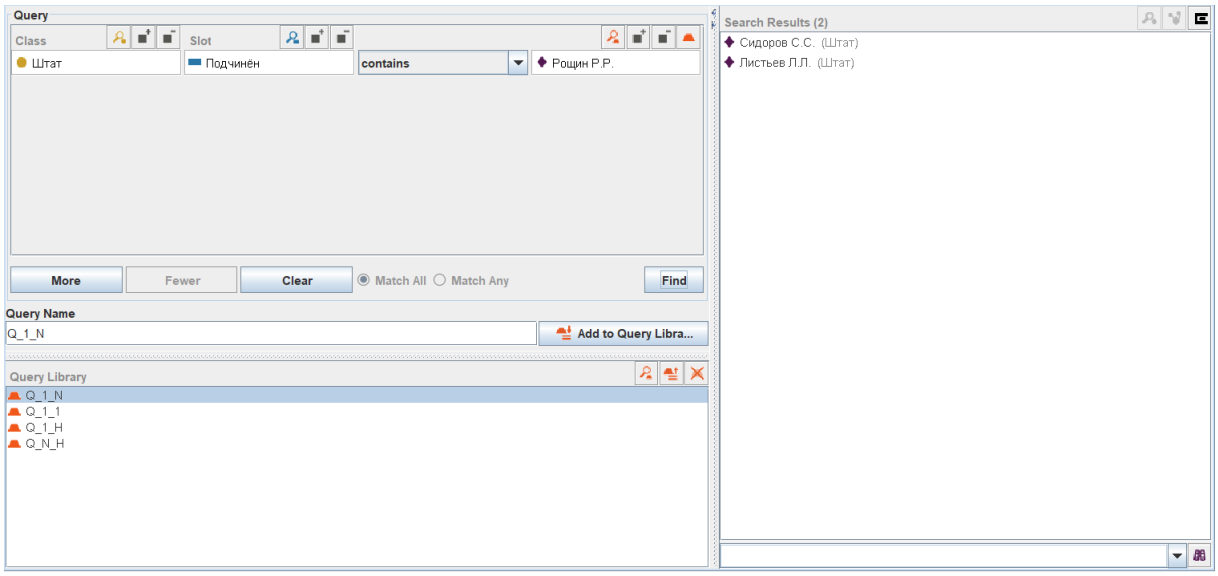


Рисунок 1.9 – Запрос один ко многим

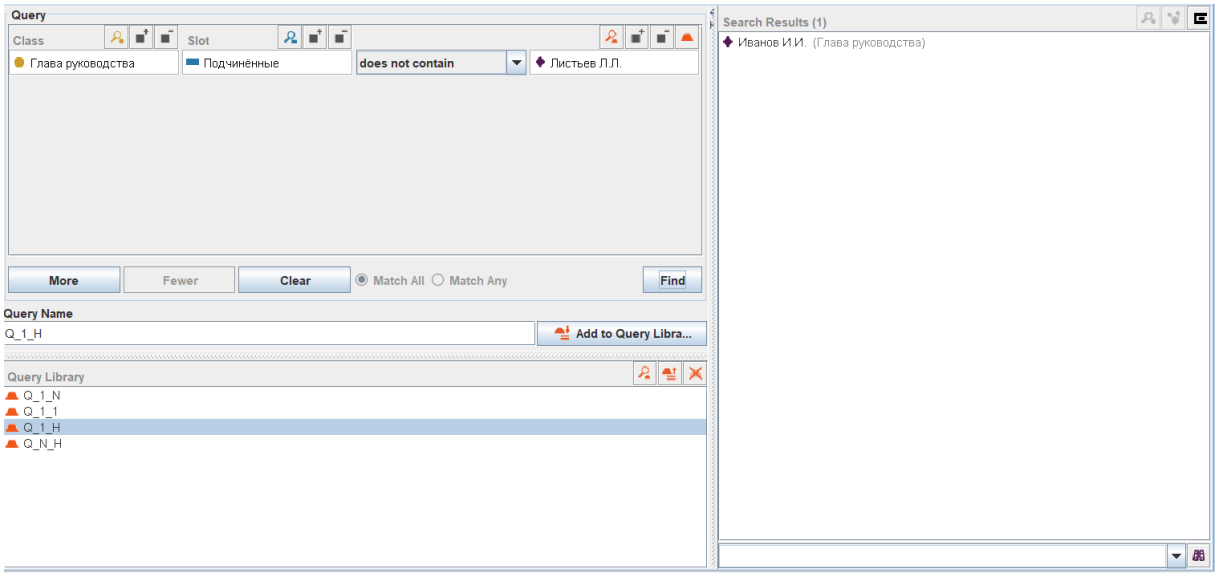


Рисунок 1.10 – Запрос один к одному через рукопожатие

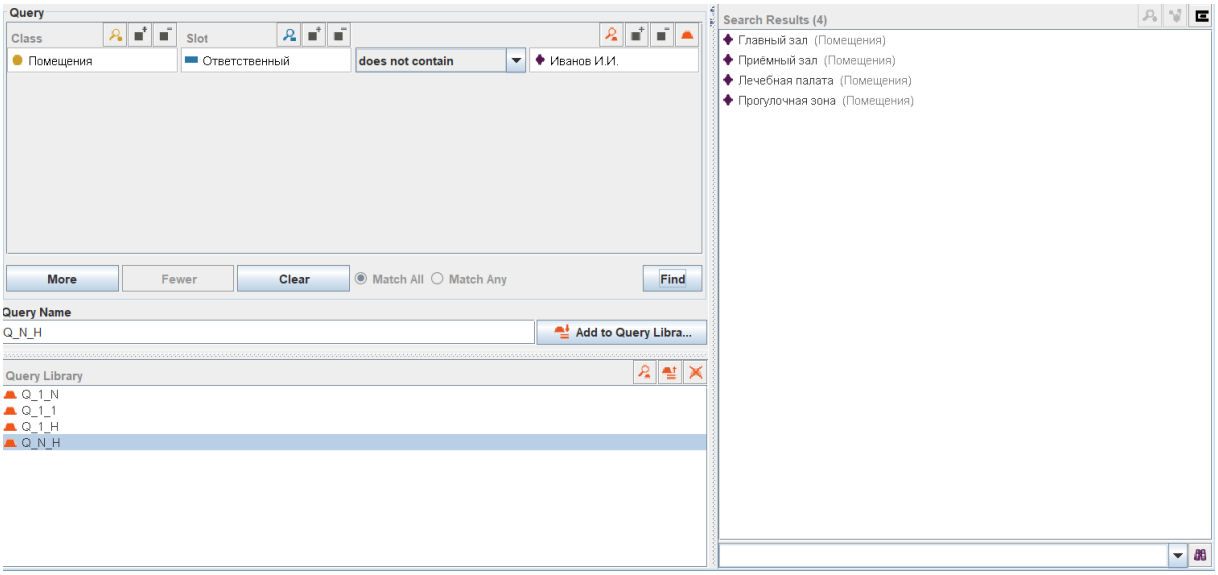


Рисунок 1.11 – Запрос один ко многим через рукопожатие

1.5 Разработка базы знаний в кодовом виде

Перенесем базу знаний в программный вид, выполним реализацию на языке Python. Реализация представлена в приложении А.

Проверим написанную базу знаний через запросы.

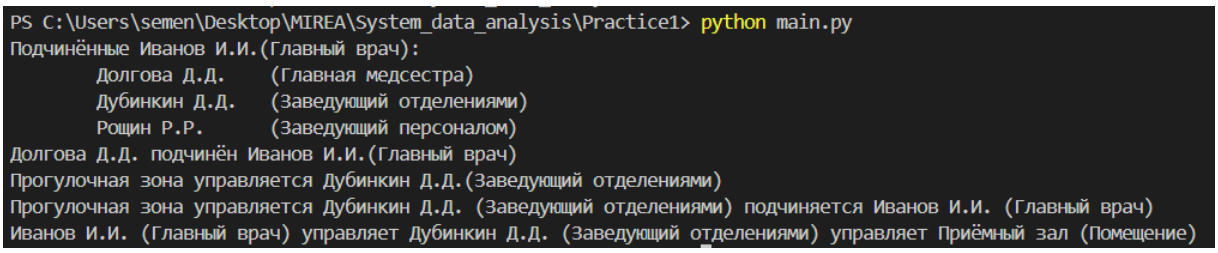


Рисунок 1.12 – Пример выполнения запросов объектов программного кода

1.6 Вывод по онтологии

В ходе выполнения практической работы была реализована база знаний по предметной области «Психиатрическая больница».

1. Проектирование модели. На основе системного анализа построена иерархическая схема, включающая руководство, сотрудников и помещения учреждения, с отображением связей подчиненности.

2. Перенос в среду Protégé. База знаний была формализована с использованием онтологического редактора Protégé. В онтологии определены классы, экземпляры и отношения, что обеспечило структурированное представление информации о предметной области.

3. Реализация на языке Python. Построенная база знаний была дополнительно реализована кодово. В программе созданы структуры данных для хранения объектов и связей между ними, а также реализованы запросы, позволяющие получать сведения о подчинённых, руководителях и закрепленных помещениях.

4. Проверка работоспособности. Тестирование программы подтвердило корректность работы базы знаний: система успешно возвращала информацию об объектах и их отношениях, что соответствует построенной модели.

Таким образом, результатом работы стала разработанная и реализованная база знаний, представленная как в визуальной форме (схема), так и в цифровой (онтология в Protégé и программная реализация на Python).

2 МЕТОД ОТЖИГА

Метод отжига служит для поиска глобального минимума некоторой функции $f(x)$, заданной для x некоторого пространства S, дискретного или непрерывного. Элементы множества S представляют собой состояние воображаемой физической системы («энергетические уровни), а значения функции f в этих точках используется как энергия системы $E = f(x)$. В каждый момент предполагается заданная температура системы T, как правило, уменьшающая с течением времени. После попадания в состояние x при температуре T, следующее состояние системы выбирается в соответствии с заданным порождающим семейством вероятностных распределений $\mathcal{G}(x, T)$, которое при фиксированных x и T задает случайный элемент $G(x, T)$ со значениями в пространстве S. После генерации нового состояния $x' = G(x, T)$, система с вероятностью $h(\Delta E, T)$ переходит к следующему состоянию x', в противном случае процесс генерации x' повторяется.

Пусть требуется найти минимум функции. В методе отжига новое состояние x_{i+1} формируется как показано в формуле 2.1.

$$x_{i+1} = x_i + T_i * \xi \tag{2.1}$$

где T_i — температура на i-ой итерации, а случайная величина ξ распределена по закону Коши. Переход в новое состояние принимается с вероятностью, которая рассчитывается по формуле 2.2.

$$p(\xi) = \frac{1}{\pi(1+\xi^2)} \tag{2.2}$$

И для изменения температуры используется формула 2.3

$$T(k) = \frac{T_0}{k} \tag{2.3}$$

Данный алгоритм применяется и для задачи Коммивояжера с использованием вышеуказанных формул.

2.1 Цель и задачи практической работы

Целью практической работы является освоение метода отжига Коши и его применение для решения задач оптимизации различного типа. В рамках работы необходимо изучить принципы работы алгоритма, особенности распределения Коши и их влияние на процесс поиска минимума.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Реализовать алгоритм отжига Коши для нахождения минимума заданной функции.
2. Применить метод отжига Коши для решения задачи коммивояжёра.
3. Проанализировать влияние параметров алгоритма на качество и скорость сходимости.
4. Сравнить полученные результаты с классическим методом отжига.
5. Сделать выводы о применимости метода отжига Коши для задач непрерывной и комбинаторной оптимизации.

2.2 Постановка задачи

В рамках практической работы необходимо реализовать нахождение минимума функции с помощью метода Отжиг Коши, также метода Отжига необходимо реализовать решение задачи Коммивояжера.

Как тестовая функция была выбрана функция Била (Рис 2.1).

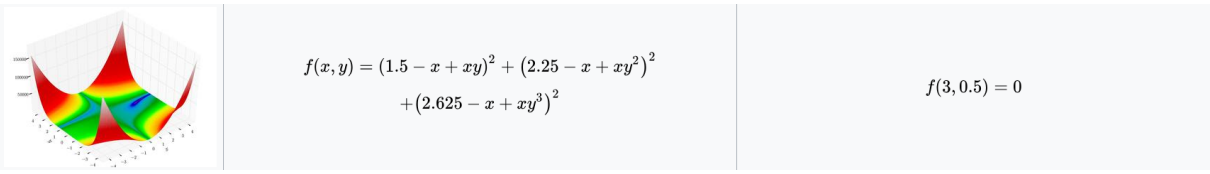


Рисунок 2.1 – Функция Била

2.3 Ручной расчёт

Выполним ручной расчёт одной итерации, для функции Била диапазон значений для *x* и *y* равен [–4.5; 4.5]. За максимальную температуру берём 1, за минимальную берём значение $1e^{-100}$.

Возьмем начальную точку: $x_b = 1.0$, $y_b = 1.0$;

Тогда значение функции *f(1,1) = 14.203125.*

Возьмём новую точку со случайными значениями.

Координаты новой точки: $x_i = 2.0$, $y_i = 0.5$;

Тогда значение функции *f(2,0,5) = 1.578125.*

Теперь необходимо сравнить значения, так как *f(1,1) > f(2,0,5),* то к *b* значениям *x* и *y* мы присваиваем у значения, после чего меняем температуру по формуле Коши, то есть делим текущую температуру на номер итерации.

Если бы, оказалось что *f(1,1) < f(2,0,5),* то смотрим вероятность $p = e^{-\Delta E/t}$, где ΔE разница между значений наших функций, а t, текущая температура, с этой вероятностью мы принимаем новые значения.

Если *f(1,1) < f(2,0,5)* и вероятность p не прошла, мы не принимаем новые значения и начинаем новые итерации.

Также проведём ручной расчёт задачи Коммивояжера, для начала составим матрицу, которая будет отображать граф.

Таблица 2.1 – Табличное представление Гамильтонова графа

	a	b	c	d	e
a		1	4	5	5
b	1		5	1	3
c	4	5		5	3
d	5	1	5		2
e	5	3	3	2	

За начальную точку возьмем вершину a.

Для начала пройдем путем a→b→c→d→e→a, посчитаем длину данного пути, 1+5+5+2+5 = 18, это путь первого пути, сохраняем его.

По аналогии с методом отжига для функции, берём максимальную температуру 1, за минимальную берём значение $1e^{-100}$, но в этом случае температура будет меняться линейно, все время умножаясь константу.

Сделаем следующую итерацию a→c→e→b→d→a, посчитаем длину этого пути 4+3+3+1+5 = 16, данный путь короче предыдущего, следовательно мы запоминаем этот путь как кратчайший путь.

Если бы, данный путь был больше, чем текущий, то с вероятностью p = $e^{-\Delta E/t}$, где ΔE разница между значений наших функций, а t, текущая температура, мы запоминаем наше значение, в противном случае переходим к следующей итерации.

2.4 Результат работы метода отжига

Реализуем нахождение минимума функции с помощью функции отжига (Рис 2.2), для примера реализованы два вида отжига, выполним реализацию на языке Python[1]. Реализация представлена в приложении Б.

```
(venv) PS C:\Users\semen\Desktop\MIREA\System_data_analysis\Practice2> py .\otzhig.py
Эталон: f(3,0.5) = 0
Отжиг
3.935875915185397 0.6474627529874599
0.0727892065728463
Отжиг Коши
2.7181819354219057 0.41515581389134404
0.018313150368945187
```

Рисунок 2.2 – Пример нахождение минимума методом отжига

2.5 Решение задачи Коммивояжера

Реализуем решение задачи Коммивояжера с помощью метода отжига(Рис 2.3), выполним реализацию на языке Python. Реализация представлена в приложении Б1.

```
(venv) PS C:\Users\semen\Desktop\MIREA\System_data_analysis\Practice2> py .\gamGraph.py
1 -> 6 -> 3 -> 2 -> 5 -> 4 -> 1
Вес: 22
1 -> 5 -> 6 -> 3 -> 4 -> 2 -> 1
Вес: 15
1 -> 2 -> 6 -> 4 -> 5 -> 3 -> 1
Вес: 15
1 -> 3 -> 6 -> 4 -> 2 -> 5 -> 1
Вес: 16
1 -> 2 -> 3 -> 6 -> 5 -> 4 -> 1
Вес: 21
1 -> 6 -> 2 -> 3 -> 4 -> 5 -> 1
Вес: 17
1 -> 4 -> 2 -> 3 -> 6 -> 5 -> 1
Вес: 16
1 -> 5 -> 6 -> 2 -> 4 -> 3 -> 1
Вес: 14
1 -> 3 -> 5 -> 2 -> 4 -> 6 -> 1
Вес: 14
1 -> 6 -> 4 -> 5 -> 3 -> 2 -> 1
Вес: 11
1 -> 2 -> 6 -> 5 -> 3 -> 4 -> 1
Вес: 14
1 -> 2 -> 4 -> 5 -> 6 -> 3 -> 1
Вес: 17
1 -> 6 -> 4 -> 3 -> 2 -> 5 -> 1
Вес: 20
1 -> 2 -> 3 -> 4 -> 6 -> 5 -> 1
Вес: 19

Кратчайший путь:
1 -> 6 -> 4 -> 5 -> 3 -> 2 -> 1
Вес: 11
```

Рисунок 2.3 – Пример решения задачи Коммивояжера

2.6 Вывод по «Метод отжига»

В ходе практической работы был выполнен ручной расчёт одной итерации метода отжига. Также ручной расчёт одной итерации задачи Коммивояжёра.

Данный расчёт наглядно показывает принцип работы метода: переход к новой точке осуществляется, если значение функции уменьшается, при этом возможен переход к худшей точке с определенной вероятностью.

Далее была выполнена кодовая реализация метода отжига и метода отжига Коши, которая позволила:

1. Автоматически находить минимум функции.
2. Решать задачу коммивояжера, минимизируя суммарное расстояние маршрута.
3. Сравнивать эффективность обычного отжига и отжига Коши, показывая, что модификация Коши обеспечивает более широкий поиск и снижает риск застревания в локальных минимумах.

В результате работы показано, что методы отжига являются эффективными инструментами как для задач непрерывной, так и для комбинаторной оптимизации.

3 РОЕВОЙ АЛГОРИТМ

РА использует рой частиц, где каждая частица представляет потенциальное решение проблемы. Поведение частицы в гиперпространстве поиска решения все время подстраивается в соответствии со своим опытом и опытом своих соседей. Кроме этого, каждая частица помнит свою лучшую позицию с достигнутым локальным лучшим значением целевой (фитнесс-) функции и знает наилучшую позицию частиц - своих соседей, где достигнут глобальный на текущий момент оптимум. В процессе поиска частицы роя обмениваются информацией о достигнутых лучших результатах и изменяют свои позиции и скорости по определенным правилам на основе имеющейся на текущий момент информации о локальных и глобальных достижениях. При этом глобальный лучший результат известен всем частицам и немедленно корректируется в том случае, когда некоторая частица роя находит лучшую позицию с результатом, превосходящим текущий глобальный оптимум. Каждая частица сохраняет значения координат своей траектории с соответствующими лучшими значениями целевой функции, которые обозначим $\boldsymbol{y_i}$, которая отражает когнитивную компоненту. Аналогично значение глобального оптимума, достигнутого частицами роя, будем обозначать $\boldsymbol{\widehat{y_i}}$, которое отражает социальную компоненту. Таким образом, каждая частица роя подчиняется достаточно простым правилам поведения (изложенным ниже формально), которые учитывают локальный успех каждой особи и глобальный оптимум всех особей (или некоторого множества соседей) роя.

Скорость будет рассчитываться по формуле 3.1.

$$v_i(t+1) = v_i(t) + c_1 * r_1 * (y_{besti} - x_i) + c_2 * r_2 * (y_{best} - x_i) \quad (3.1)$$

Где y_{besti} лучшее значение данной точки, y_{best} лучшее значение точки всего роя. c_1 и c_2 константы, r_1 и r_2 случайные числа из диапазона [0;1].

В нашем случае за константы с возьмём значение 2, а для констант r возьмём значение 0,3 и 0,7 соответственно.

3.1 Цель и задачи практической работы

Целью практической работы является изучение принципов роевого алгоритма и его применение для решения задач оптимизации. В рамках работы необходимо освоить механику обновления положения и скорости частиц, а также влияние обмена информацией между ними на поиск оптимального решения.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Выполнить ручной расчёт одной итерации роевого алгоритма для заданной функции, определяя новые координаты частиц и значения функции.
2. Реализовать роевой алгоритм на языке Python для автоматического поиска минимума функции.
3. Применить алгоритм для решения задачи коммивояжёра и минимизации суммарного расстояния маршрута.
4. Проанализировать влияние параметров алгоритма (скорости, веса инерции, коэффициенты притяжения) на качество и скорость сходимости.
5. Сравнить эффективность результатов ручного расчёта и программной реализации, выявив преимущества алгоритма для задач оптимизации.

3.2 Постановка задачи

В рамках практической работы необходимо реализовать роевый алгоритм вручную и кодово, на примере нахождения минимум функции.

Как тестовая функция была выбрана функция Била (Рисунок 3.1).

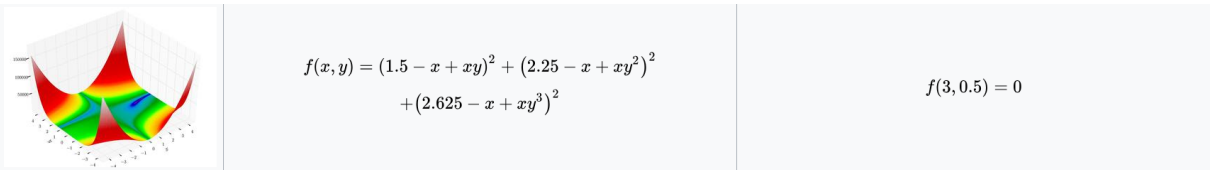


Рисунок 3.1 – Функция Била

3.3 Ручной расчёт

Выполним ручной расчёт одной итерации, для функции Била диапазон значений для *x* и *y* равен [–4.5; 4.5].

Проинициализируем 4 единицы роя.

Таблица 3.1 – Начальные значения роя

	x	y	f(x,y)
1	-2.9	0.8	105.28
2	0.17	-3.16	25.56
3	-1.93	-3.29	5764.5
4	-4.02	-1.25	319.49

В данной таблице лучшее значение имеет элемент 2, следовательно y_{best} будет иметь значение 2-ого элемента.

Далее вычисляем скорость для каждой координаты частицы, после чего пересчитаем новые координаты точек.

Таблица 3.2 – Значения скорости роя и новые координаты точек

	V_x	V_y	x	y
1	4,30	-5,54	1,40	-4,44
2	0	0	0,17	-3,16
3	2,94	0,18	1,01	-3,11
4	5,87	-2,67	1,85	-3,92

Далее считаем значение функций, для новых координат, находим новый глобальный минимум и локальный минимум для каждой точки, после чего повторяем итерацию с вычислением скорости.

3.4 Результат работы метода отжига

Реализуем нахождение минимума функции с помощью роевого алгоритма (Рис. 3.2), количество частиц: 5, количество итераций: 100, выполним реализацию на языке Python[1]. Реализация представлена в приложении В.

```
(venv) PS C:\Users\semen\Desktop\MIREA\System_data_analysis\Practice3> py .\roi.py
==Начальные позиции роя==
x = -2.8118844235287472  y = 0.6503402048544427  f(x,y) = 42.911313269402704
x = -1.6323314149361305  y = -3.2379058136401273  f(x,y) = 3806.304456246304
x = 4.239373855721958  y = -1.2807597673281554  f(x,y) = 202.06812705028628
x = 0.5477258211238265  y = -2.967442193094229  f(x,y) = 192.73007708957925
x = -1.285295448870194  y = 2.40605701782626  f(x,y) = 211.13659044284802
(-2.8118844235287472, 0.6503402048544427, 42.911313269402704)

==Конечные позиции роя==
x = -0.6329231209511166  y = 0.9136962440648588  f(x,y) = 15.662055458007865
x = 1.2881768126020332  y = -0.7880469042762901  f(x,y) = 4.248279512323005
x = -1.4751303862013527  y = -0.5514361538969439  f(x,y) = 43.98978501660606
x = 3.4100334073391894  y = 3.179937439243254  f(x,y) = 13042.07709137118
x = -0.9585780146181655  y = 0.3023480747995798  f(x,y) = 27.096669324623512
(3.0641578414285267, 0.5145068757988449, 0.0006384524359516067)
```

Рисунок 3.2 – Пример выполнения роевого алгоритма

3.5 Вывод по роевому алгоритму

В ходе практической работы была выполнена одна итерация ручного расчёта роевого алгоритма для двух частиц.

Данный расчёт демонстрирует работу роевого алгоритма: частицы корректируют свои скорости и позиции, ориентируясь на личный и глобальный опыт, постепенно приближаясь к оптимальному значению.

Далее была выполнена кодовая реализация роевого алгоритма, которая позволила:

1. Автоматически находить минимум функции в многомерном пространстве.
2. Применять алгоритм для решения задачи коммивояжёра, минимизируя суммарное расстояние маршрута.
3. Анализировать влияние параметров алгоритма (веса инерции, коэффициенты притяжения) на скорость сходимости и точность результата.

В результате работы показано, что роевой алгоритм является эффективным методом как для непрерывной, так и для комбинаторной оптимизации, обеспечивая скоординированный поиск оптимальных решений.

4 МУРАВЬИНЫЙ АЛГОРИТМ

Первый муравьиный алгоритм был разработан М.Дориго. По современной классификации он относится к (**antsystem**) муравьиной системе (МС). По сравнению с простым муравьиным алгоритмом в МС улучшены характеристики за счет изменения метода вычисления вероятности выбора следующей вершины путем учета эвристической информации и ввода списка запрещенных вершин (tabulist). Конкретно, в МС вероятность перехода из -ой вершины в j-ю вершину определяется следующим образом.

Перемещение по графу основывается на вероятности, с которой выбирается следующая вершина и рассчитывается по формуле:

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\tau_{ij}^{\alpha} * \mu_{ij}^{\beta}}{\sum_{j \in N_i^k} \tau_{ij}^{\alpha} * \mu_{ij}^{\beta}}, \text{если } j \in N_i^k \\ 0, \text{в ином случае} \end{cases} \quad (4.1)$$

Здесь N_i^k представляет множество возможных вершин, связанных с -й вершиной, для k-го муравья. Если для любого i-го узла и k-го муравья $N_i^k = \emptyset$, тогда предшественник узла i включается в N_i^k. В этом случае в пути возможны петли. Когда все муравьи построили полный путь от начальной до конечной вершины, удаляются петли в путях, и каждый муравей помечает свой построенный путь, откладывая для каждой дуги феромон в соответствии со следующей формулой.

$$\Delta\tau_{ij}^{k}(t) = \frac{q}{L^k(t)} \quad (4.2)$$

Здесь $L^k(t)$ – длина пути, построенного k-м муравьем в момент времени t.

Таким образом, для каждой дуги графа концентрация феромона определяется следующим образом:

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \sum_{k=1}^{n_k} \Delta\tau_{ij}^{k}(t) \quad (4.3)$$

где n_k - число муравьев.

4.1 Цель и задачи практической работы

Целью практической работы является изучение принципов муравьиного алгоритма и его применение для решения задач нахождения кратчайшего пути. В рамках работы необходимо освоить механику работы феромонов, которые помогают найти лучший путь

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Выполнить ручной расчёт одной итерации муравьиного алгоритма для гамильтонова графа.
2. Реализовать муравьиный алгоритм на языке Python для автоматического поиска кратчайшего пути.
3. Применить алгоритм для решения задачи коммивояжёра.
4. Сравнить эффективность результатов ручного расчёта и программной реализации, выявив преимущества алгоритма для задач оптимизации.

4.2 Постановка задачи

В рамках практической работы необходимо реализовать роевый алгоритм вручную и кодово, алгоритм будет проверятся на задаче коммивояжёра где граф состоит из шести вершин.

4.3 Ручной расчёт

Выполним ручной расчёт одной итерации, для гамильтонова графа.

Построим матрицу, которая будет описывать граф для ручного расчёта.

Таблица 4.1 – Значения матрицы

	1	2	3	4	5
1		10	15	20	25
2	10		35	25	30
3	15	35		30	10
4	20	25	30		15
5	25	30	10	15	

Исходное количество муравьев: 3.

Начальные феромоны для всех ребер: 0.2.

Начальная вершина: 1.

Рассчитаем вероятность перехода из первой вершины в каждую Вершину, для этого используем формулу 4.1:

$P_{10} \approx 0.002$

$P_{12} \approx 0.06035$

$P_{13} \approx 0.1183$

$P_{14} \approx 0.0820$

Сделаем выбор для муравьев:

Муравей 1: переходит в 0

Муравей 2: переходит в 3

Муравей 3: переходит в 4

Далее рассмотрим полный путь только одного муравья, текущий путь которого равен $1 \rightarrow 0$.

По формуле 4.1 рассчитаем вероятность для следующего перехода:

$P_{02} \approx 0.5200$

$P_{03} \approx 0.2926$

$P_{04} \approx 0.1874$

Переходим в 3-ю вершину, рассчитаем вероятности для следующих вершин, на данный момент путь муравья $1 \rightarrow 0 \rightarrow 3$.

P_{32}=0.2

P_{34}=0.8

Переходим в 4-ую вершину. Далее переходим во вторую, так как она осталась последней, и замыкаем цикл, в итоге первый муравей имеет путь 1 →0 →3 →4 →2 →1 длинна которого 90.

По формуле 4.2 получаем что для каждого ребра, который прошёл первый муравей добавляется 1.112 феромоны, добавление идёт после того как все муравьи пройдут свой путь и рассчитается по формуле 4.3.

Проходы по ребрам для остальных муравьев остаётся идентичным, поэтому не рассматриваются.

При следующих итерациях данные шаги будут повторяться и муравьи будут находить наилучшее решение задачи Коммивояжёра.

4.4 Результат работы

Реализуем нахождение кратчайшего пути с помощью муравьиного алгоритма, количество муравьев: 10, количество итераций: 100, выполним реализацию на языке Python[2]. Реализация представлена в приложении Г.

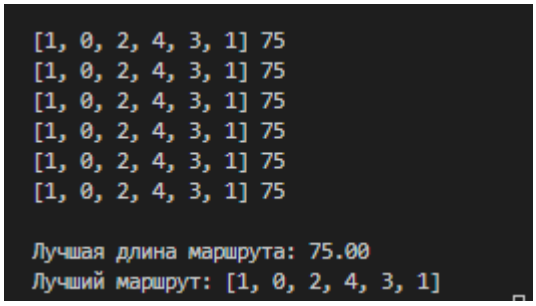


Рисунок 4.1 – Пример выполнения муравьиного алгоритма

4.5 Вывод по муравьиному алгоритму

В ходе практической работы был выполнен ручной расчёт одной итерации муравьиного алгоритма.

Данный расчёт демонстрирует работу муравьиного алгоритма: муравьи находят кратчайший путь, основываясь на ферамонах, которые оставляют другие муравьи прошедшие эти пути.

Далее была выполнена кодовая реализация данного алгоритма, которая позволила:

1. Автоматически находить кратчайший путь.
2. Применять алгоритм для решения задачи коммивояжёра.

В результате работы показано, что муравьиный алгоритм является эффективным методом для нахождения кратчайшего пути и может использоваться в многих практических задачах использующих графы.

5 ПЧЕЛИНЫЙ АЛГОРИТМ

Алгоритм основан на поведении роя пчёл, и представляет из себя стаю разведчиков, которые изначально находят лучшие точки, и потом в окрестности этих точек посылаются рабочие пчёлы которые продолжают искать точки экстремума.

5.1 Цель и задачи практической работы

Целью практической работы является изучение принципов пчелиного алгоритма и его применение для решения задач нахождения точек экстремума.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1 Выполнить ручной расчёт одной итерации пчелиный алгоритма для функции Била.

2 Реализовать пчелиный алгоритм на языке Python для автоматического поиска точек экстремума.

3 Применить алгоритм для функции Била(Таблица 5.1).

Таблица 5.1 – Функция Била

Формула	Глобальный минимум	Метод поиска
$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$	f(3, 0.5) = 0	$-4.5 \leq x,y \leq 4.5$

5.2 Постановка задачи

В рамках практической работы необходимо реализовать пчелиный алгоритм вручную и кодово, алгоритм будет проверятся на функции Била.

В начале алгоритма в точки, описываемые случайными координатами, отправляется некоторое количество пчел-разведчиков (пусть будет S пчел, от слова scout). Таким образом:

1 шаг: Необходимо задать количество пчел-разведчиков S. В точки со случайными координатами $X_{\beta,0} \in D$, отправляются пчелы-разведчики, где β – номер пчелы разведчика, $\beta \in [1: S]$, а 0 обозначает номер итерации в данный момент времени. Считаются значения целевой функции $F(X)$ в этих точках.

2 шаг: В области D с помощью полученных значений выделяют два вида участков (подобластей) d_β .

Первый вид содержит n лучших участков, которые соответствуют наибольшим или наименьшим значениям целевой функции, в зависимости от того решается задача на минимум или на максимум функции.

Второй m перспективных участков, соответствующих значениям целевой функции, наиболее близким к наилучшим значениям.

Подобласть d_β является подобластью локального поиска, представляющая собой гиперкуб в пространстве R^k с центром в точке $X_{\beta,0}$. Длина его сторон равна 2Δ , где Δ – параметр, называемый размером области локального поиска.

3 шаг: Сравнивается евклидово расстояние $\left\|X_{\beta,0} - X_{\gamma,0}\right\|$ между двумя агентами-разведчиками. Для точек $A = (x_1, x_2, \dots, x_n)$ и $B = (y_1, y_2, \dots, y_n)$ евклидово расстояние считается по формуле 5.1.

$$d(A,B) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} \qquad (5.1)$$

Если евклидово расстояние оказывается меньше фиксированной величины, то возможны два следующих варианта метода :

• поставить в соответствие этим агентам два различных пересекающихся участка d_β, d_γ (лучших и/или перспективных);

• поставить в соответствие тем же агентам один участок, центр которого находится в точке, соответствующей агенту с большим значением целевой функции. Из этих двух вариантов в работе используется второй вариант.

4 шаг: В каждый из лучших и перспективных участков посылается по N и по M агентов, соответственно. Координаты этих агентов в указанных участках определяются случайным образом.

5 шаг: В полученных точках снова считается значение целевой функции $F(X)$, снова выбирается наибольшее или наименьшее значение. Точка, в которой значение функции является максимальным, становится центром новой подобласти.

6 шаг: Шаги 4 и 5 повторяются до тех пор, пока не будет получен искомый результат, если такой известен, либо до тех пор, пока полученные значения координат экстремумов и значений функции в них не повторятся τ раз, где τ — параметр останова.

5.3 Ручной расчёт

Выполним ручной расчёт одной итерации данного алгоритма на примере функции Била.

Первоначально на случайные точки отправляются пчёлы разведчики. Из всех точек, на которое попали пчёл разведчики выбирается n лучших точек и m перспективных точек.

Рассмотрим пчелу, которая попала на координаты (2, 1), значения функции в данной точке будет равно 14.203125. Для дальнейшего расчёта будем воспринимать эту точку как лучшее значение в рое.

Теперь в область этой точки отправим рой рабочих пчёл.

Для первой координаты [2 – 1 = 1; 2 + 1 = 3]

Для второй координаты [1 – 1 = 0; 1 + 1 = 2]

В эту область отправляем рабочих пчёл, и рассчитываем значение функции в точках рабочих пчёл, данные действия производятся для всех найденных лучших и перспективных областей, после чего среди всех новых точек снова отмечаются лучшие, а процесс повторяется.

5.4 Результат работы

Реализуем нахождение точек экстремума с помощью пчелиного алгоритма, количество пчёл разведчиков: 10, пчёл отправляющихся в лучшие области 5, лучших областей 2 количество итераций: 1000, выполним реализацию на языке Python[3]. Реализация представлена в приложении Д.

```
(venv) PS C:\Users\semen\Desktop\MIREA\System_data_analysis\Practice5> py .\pchela.py
14.203125
Эталон: f(3, 0.5) = 0
Результат x = 3.01150 y = 0.50096 f = 0.00010
```

Рисунок 5.1 – Пример выполнения пчелиного алгоритма

5.5 Вывод по пчелиному алгоритму

В ходе практической работы был выполнен ручной расчёт одной итерации пчелиного алгоритма.

Данный расчёт демонстрирует работу пчелиного алгоритма: пчелы разведчики находят лучшие и перспективные точки, на которые отправляются рабочие пчёлы и происходит точек экстремума функции.

Далее была выполнена кодовая реализация данного алгоритма, которая позволила автоматически находить точки минимума функции;

В результате работы показано, что пчелиный алгоритм является эффективным методом для точек экстремума функции и может использоваться в задачах оптимизации.

6 АЛГОРИТМ ОБЕЗЬЯН

Алгоритм основан на поведении стаи обезьян, они перемещаются с помощью прыжков в окрестностях своего местоположения и если не находят лучшего значения, то перемещаются к центру нахождения всех обезьян, в следствие чего находят точки экстремума функций.

Первоначально все обезьяны отправляются на случайные точки.

После каждая обезьяна начинает делать локальные прыжки фиксированное количество раз, если после прыжка, значение функции меньше изначального, количество прыжков обнуляется.

$$x_{ij} = ((x_{ij} - b); (x_{ij} + b)) \tag{6.1}$$

Формула 6.1, где b – длинна прыжка, используется вычисления длинны прыжка.

Если лимит прыжков исчерпан, обезьяна совершает глобальный прыжок в сторону центра всех обезьян, перелетая его.

$$x_{ij} = x_{ij} + \gamma(x_j^c - x_{ij}) \tag{6.2}$$

$$x_j^c = \frac{1}{S}\sum_{i=1}^{S} x_{ij} \tag{6.3}$$

Формула 6.2 является экстраполяционной функцией, которая рассчитывает куда производится глобальный прыжок, x_j^c– центральная точка среди всех обезьян, получаемая по формуле 6.3, где S – количество обезьян, x_{ij} – текущее положение обезьяны. γ – случайный параметр в диапазоне от 1 до 2, с помощью которого обезьяна перелетает центр и данные формулы вычисляются как для первой так и для второй координаты.

Данные действия повторяются для всей стаи обезьян пока не будет выполнен критерий остановки.

6.1 Цель и задачи практической работы

Целью практической работы является изучение принципов алгоритма обезьян и его применение для решения задач нахождения точек экстремума.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Выполнить ручной расчёт одной итерации алгоритма обезьян для функции Била.
2. Реализовать алгоритм обезьян на языке Python для автоматического поиска точек экстремума.
3. Применить алгоритм для функции Била.

Таблица 6.1 – Функция Била

Формула	Глобальный минимум	Метод поиска
$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$	f(3, 0.5) = 0	$-4.5 \leq x,y \leq 4.5$

6.2 Постановка задачи

В рамках практической работы необходимо реализовать алгоритм обезьян вручную и кодово, алгоритм будет проверятся на функции Била.

6.3 Ручной расчёт

Выполним ручной расчёт одной итерации данного алгоритма на примере функции Била.

Первоначально все обезьяны отправляются на случайные точки.

После каждая обезьяна начинает делать локальные прыжки фиксированное количество раз, если после прыжка, значение функции меньше изначального, количество прыжков обнуляется.

Допустим, что обезьяна попала в точку (2, 1) со значением функции 14.203125, далее необходимо определить область, в которой обезьяна будет совершать прыжки.

За *b* возьмём 1. По формуле 6.1 вычислим область для первой и для второй координаты.

Для первой координаты [2 – 1 = 1; 2 + 1 = 3]

Для второй координаты [1 – 1 = 0; 1 + 1 = 2]

Перейдём к ручному расчёту одной итерации, возьмём 3 обезьяны с максимальным количеством локальных прыжков равных 5.

Обезьяна 1: x = 1.0, y = 2.0, f(x,y) = 126.45;

Обезьяна 1: x = 2.0, y = 0.5, f(x,y) = 39.45;

Обезьяна 1: x = 3.0, y = 1.0, f(x,y) = 14.20;

По формуле 6.3 рассчитаем центр стаи.

Центр: $x_c = 0.67$ и $y_c = 1.17$;

Совершим локальные прыжки для первой обезьяны, область прыжка будет рассчитываться по формуле 6.1.

Обезьяна 1:

Прыжок 1: Область прыжка: [0.9, 1.1] и [1.9, 2.1];

Новое место: x = 1.05, y = 1.95, f(x,y) = 120.82;

Это значение лучше, сбрасываем количество оставшихся прыжков до 5 и записываем эти значения в лучший результат.

Прыжок 2: Область прыжка: [0.95, 1.15] и [1.85, 2.05];

Новое место: x = 1.12, y = 1.88, f(x,y) = 112.02;

Это значение лучше, сбрасываем количество оставшихся прыжков до 5 и записываем эти значения в лучший результат.

Прыжок 3: Область прыжка: [1.02, 1.22] и [1.78, 1.98];

Новое место: x = 1.18, y = 1.82, f(x,y) = 103.48;

Это значение лучше, сбрасываем количество оставшихся прыжков до 5 и записываем эти значения в лучший результат.

Прыжок 4: Область прыжка: [1.08, 1.28] и [1.72, 1.92];

Новое место: x = 1.25, y = 1.75, f(x,y) = 94.44;

Это значение лучше, сбрасываем количество оставшихся прыжков до 5 и записываем эти значения в лучший результат.

Прыжок 5: Область прыжка: [1.15, 1.35] и [1.65, 1.85];

Новое место: x = 1.30, y = 1.70, f(x,y) = 87.177;

Это значение лучше, сбрасываем количество оставшихся прыжков до 5 и записываем эти значения в лучший результат.

Данные шаги повторяются, пока не случится 5 прыжков, при которых новое значение функции будет хуже прошлых. Если за всё количество локальных прыжков значение функции не улучшилось, обезьяна совершает глобальный прыжок, иначе остаётся в своей точке. Центр стаи у нас есть это координаты x_c и y_c.

Воспроизведём ситуацию, при которой первая обезьяна постоянно совершала бы плохие прыжки и ей пришлось бы делать глобальный прыжок, воспользуемся формулой 6.2 и рассчитаем координаты прыжка.

Тогда обезьяна 1 находится в точке (1,2), центр находится в точке (0.67,1.17) и γ равен 1.5.

Новые значения : x = 0.505 y = 0.921, данные значения находятся в укзанных пределах, поэтому рассчитаем функцию f(0.505, 0.921) = 13.174.

После расчёта новых координат первой обезьяны, мы переходим к следующей и так пока все обезьяны не перейдут к новым позициям.

6.4 Результат работы

Реализуем нахождение точек экстремума с помощью алгоритма обезьян, количество обезьян: 15, максимальное количество локальных прыжков: 20, шаг 0.1, количество итераций: 1000, выполним реализацию на языке Python[3] (Рис. 6.1). Реализация представлена в приложении Е.

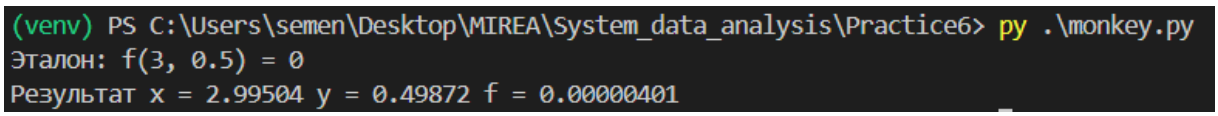


Рисунок 6.1 – Пример выполнения алгоритма обезьян

6.5 Вывод по «Алгоритм обезьян»

В ходе практической работы был выполнен ручной расчёт одной итерации алгоритма обезьян.

Данный расчёт демонстрирует работу алгоритма обезьян: обезьяны совершают локальные и глобальные прыжки пока не будет выполнен критерий остановки.

Далее была выполнена кодовая реализация данного алгоритма, которая позволила автоматически находить точки минимума функции;

В результате работы показано, что алгоритм обезьян является эффективным методом для точек экстремума функции и может использоваться в задачах оптимизации.

7 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ

Генетические алгоритмы (ГА) – адаптивные методы поиска, которые в последнее время часто используются для решения задач функциональной оптимизации. Они основаны на генетических процессах биологических организмов: биологические популяции развиваются в течении нескольких поколений, подчиняясь законам естественного отбора и по принципу «выживает наиболее приспособленный», открытому Чарльзом Дарвином. Подражая этому процессу генетические алгоритмы способны «развивать» решения реальных задач, если те соответствующим образом закодированы.

Генетические алгоритмы являются наиболее известным и популярным методом эволюционных вычислений, т.к. используются для широкого круга задач: оптимизации, поиска, управления и т.д. Данные алгоритмы адаптивны, развивают решения, развиваются сами.

Для расчёта будут использованы следующие формулы:

Фитнес функция:

$$\text{fit(x)} = \frac{1}{1+f(x)} \tag{7.1}$$

Функция для линейной комбинации родителей при создании нового потомка:

$$\text{c} = \alpha * p_1 + (1 - \alpha) * p_2 \tag{7.2}$$

Где p – координаты точек родителей и α – случайный параметр:

Формула расчёта мутации:

$$x_i^{t+1} = x_i^t + \delta \tag{7.3}$$

Где δ – случайная величина распределённая по нормальному закону. Данная формула работает как для первой так и для второй координаты.

7.1 Цель и задачи практической работы

Целью практической работы является освоение генетического метода и его применение для решения задач оптимизации различного типа. В рамках работы необходимо изучить принципы работы алгоритма.

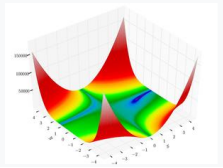
Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Реализовать генетический алгоритм для нахождения минимума заданной функции.
2. Проанализировать влияние параметров алгоритма на качество и скорость сходимости.
3. Сравнить полученные результаты с другими методами.
4. Сделать выводы о применимости генетического алгоритма для задач оптимизации.

7.2 Постановка задачи

В рамках практической работы необходимо реализовать нахождение минимума функции с помощью генетического алгоритма.

Как тестовая функция была выбрана функция Била.



$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$$

$$f(3, 0.5) = 0$$

Рисунок 7.1 – Функция Била

7.3 Ручной расчёт

Выполним ручной расчёт одной итерации, для функции Била диапазон значений для *x* и *y* равен [–4.5; 4.5].

Начнём расчёт, для начала инициализируем популяцию, назначив каждой особи координаты и для этих координат вычеслив функцию и фитнесс функцию по формуле 7.1:

Особь 1 : x = 2.1, y = -1.8, f(x,y) = 204.95, fit = 0.005;

Особь 2 : x = 0.3, y = 3.2, f(x,y) = 177.64, fit = 0.006;

Особь 3 : x = -2.7, y = -0.5, f(x,y) = 81.14, fit = 0.012;

Особь 4 : x = 1.5, y = 2.8, f(x,y) = 1333.75, fit = 0.0007;

Особь 5 : x = -3.9, y = 1.2, f(x,y) = 0.85, fit = 0.540;

Особь 6 : x = 4.2, y = -2.3, f(x,y) = 3339.16, fit = 0.0003;

Особь 7 : x = -0.8, y = 0.9, f(x,y) = 16.34, fit = 0.058;

Особь 8 : x = 3.6, y = -3.1, f(x,y) = 12993.33, fit = 0.00008;

Особь 9 : x = -1.4, y = 4.1, f(x,y) = 8955.14, fit = 0.00011;

Особь 10 : x = 2.8, y = 1.5, f(x,y) = 127.49, fit = 0.0077;

Первое поколение сформировано, теперь на его основе сформируем новое поколение, сначала производится турнирный отбор, где выбираются 3 случайные особи, и потом выбирается одна лучшая особь из этих трёх.

Родитель 1:

Пусть случайные особи: 2, 5, 9, из данных особей лучшей является особь под номером 5.

Родитель 2:

Пусть случайные особи: 3, 7, 10, из данных особей лучшей является особь под номером 7.

Следующий этап, после выбора родителей, формирование детей, с вероятностью 80% два ребёнка образуются с помощью формулы 7.2 и с вероятностью 20% оба ребёнка копируют родителей. Воспользуемся формулой 7.2 с учётом что $\alpha = 0.4$.

Ребёнок 1: x = -2.04, y = 1.02;

Ребёнок 1: x = -2.66, y = 1.08;

Произведём мутацию детей, основываясь на формуле 7.3, мутация производится с вероятностью 0.1, для каждой координаты отдельно.

Допустим что для ребёнка 1 для координаты x мутация не прошла, а для координаты y прошла, тогда высчитав мутацию по формуле 7.3 новые значения координат: x = -2.04 и y = 1.09.

Допустим что для ребёнка 2 для координаты x мутация прошла, а для координаты y не прошла, тогда высчитав мутацию по формуле 7.3 новые значения координат: x = -2.69 и y = 1.08.

Так мы повторяем этот алгоритм, пока у нас не образуется 10 детей, которые будут является вторым поколением, полностью основанном на первом поколении. После чего производится расчёт нового поколения и так заданное количество итераций.

7.4 Результат работы генетического алгоритма

Реализуем нахождение минимума функции с помощью генетического алгоритма[4]. Реализация представлена в приложении Б.

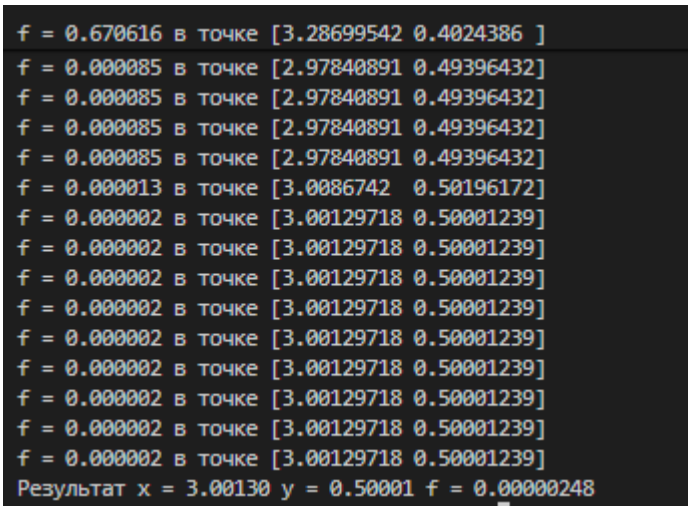


Рисунок 7.2 – Пример нахождение минимума генетическим алгоритмом

7.5 Вывод по генетическому алгоритму

В ходе практической работы был выполнен ручной расчёт одной итерации генетического алгоритма. Также ручной расчёт одной итерации задачи Коммивояжёра.

Данный расчёт наглядно показывает принцип работы метода: формирование нового поколения на основе работы старого и небольшой мутации как в нахождении минимума функции так и в решении задачи Коммивояжёра.

Далее была выполнена кодовая реализация алгоритма. В результате работы показано, что алгоритм являются достаточно эффективными инструментом как для задач оптимизации и нахождения кратчайшего пути.

ЗАКЛЮЧЕНИЕ

Завершая изучение курса «Системный анализ данных в системах поддержки принятия решений», можно отметить, что проверка рассмотренных методов на задачах нахождения минимума функций и решении задачи коммивояжёра позволила убедиться в их практической эффективности. Использование онтологических моделей дало возможность формализовать знания о предметной области и обеспечить корректную интерпретацию данных, а природоподобные алгоритмы продемонстрировали способность находить оптимальные решения в ситуациях, где традиционные подходы сталкиваются с трудностями. Метод отжига, роевые технологии, муравьиные и пчелиные алгоритмы показали, как механизмы, вдохновлённые процессами природы, позволяют успешно преодолевать сложный рельеф функции или большое число возможных маршрутов в TSP.

Проведём анализ используемых алгоритмов.

Таблица 8.1 – Анализ методов и алгоритмов

Название	Время работы	Кол-во частиц	Кол-во итераций	Результат
Метод отжига	0.003 сек	1	200	0.072
Роевой алгоритм	0.007 сек	5	100	0.042
Пчелиный алгоритм	0.092 сек	29	100	0.489
Алгоритм обезьян	0.039 сек	15	100	0.0000008
Генетический алгоритм	0.005 сек	30	100	0.006

Проанализировав все методы необходимо выделить генетический алгоритм, данный алгоритм является крайне эффективным и не уступает другим алгоритмам как в нахождении минимума функции, также является относительно

простым в реализации, так как не использует сложных формул. Но в то же время есть самый эффективный алгоритм обезьян, хоть и требует больше времени но за то же количество итераций, находит значения, которые максимально приближены к минимуму функции, но данный метод требует понимания сложных математических формул экстраполяции. Также есть метод отжига, хоть он и не находит такие же точные значения как алгоритм обезьян, но он очень прост в реализации и требует меньше всего времени работы. И данный алгоритм крайне эффективный в задаче Коммивояжера, выигрывая у муравьиного алгоритма, хотя и находит кратчайший путь за 1 секунду, когда муравьиный за 0,1, но муравьиный алгоритм требует математического понимания формул, и требует долгого анализа работы и изменения параметров для улучшения работы когда метод отжига крайне прост в реализации.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Python Software Foundation. Python Documentation — [Электронный ресурс]. URL: https://docs.python.org/3/ (дата обращения: 15.09.2025).

2. Лутц М. Изучаем Python. 5-е изд. / пер. с англ. — Санкт-Петербург: Символ-Плюс, 2019. — 1648 с.

3. Баляев С. А. Объектно-ориентированное программирование. Учебное пособие. — Москва : ФОРУМ, ИНФРА-М, 2020. — 256 с.

4. Гринберг Д. Программирование на Python 3. Подробное руководство. — Москва : Вильямс, 2014. — 832 с.

ПРИЛОЖЕНИЕ

Приложение А – Код программы Онтологиию.

Приложение Б – Код программы Метод отжига.

Приложение Б.1 – Код программы Решение задачи Коммивояжера.

Приложение В – Код программы Роевой алгоритм.

Приложение Г – Код программы Муравьиный алгоритм.

Приложение Д – Код программы Пчелиного алгоритм.

Приложение Е – Код программы Алгоритм обезьян.

Приложение Ж – Код программы Генетический алгоритм.

Приложение А

Код программы Онтологии

Листинг А.1 — Основной алгоритм программы

```
from abc import ABC, abstractmethod
class PsyHospital:
    pass
class Staff:
    pass
class HeadHosp:
    pass
class AdmHosp:
    pass
class WorkerHosp:
    pass
class RoomHosp:
    pass
class PsyHospital(ABC):
    pass
class Staff(PsyHospital):
    pass

class HeadHosp(Staff):
    def __init__(self, name: str, post: str):
        self.name = name
        self.post = post
        self.subs = []

    def print_subs(self):
        print(f"Подчинённые {self.name}({self.post}):")
        for obj in self.subs:
            print(f"\t{obj.name}\t({obj.post})")

    def find_boss(self, boss_name):
        if (self.boss.name == boss_name):
            return f"{self.boss.name} ({self.boss.post})"
        else:
            return None

    def find_staff (self, staff_name):
        for obj in self.subs:
            if (obj.name == staff_name):
                return f"{self.name} ({self.post}) управляет {obj.name}
({obj.post})"
            else:
                find_staff = obj.find_staff(staff_name)
                if (find_staff != None):
                    return f"{self.name} ({self.post}) управляет "
```

Листинг А.2 — Продолжение листинга А.1

```
+ find_staff
        return None
class AdmHosp(Staff):
    def __init__(self, name: str, post: str, boss: HeadHosp):
        self.name = name
        self.post = post
        self.boss = boss
        self.subs = []
        boss.subs.append(self)

    def print_subs(self):
        print(f"Подчинённые {self.name}({self.post}):")
        for obj in self.subs:
            print(f"\t{obj.name}\t({obj.post})")

    def print_boss(self):
        print(f"{self.name} подчинён
{self.boss.name}({self.boss.post})")

    def find_boss(self, boss_name):
        if (self.boss.name == boss_name):
            return f"{self.name} ({self.post}) подчиняется
{self.boss.name} ({self.boss.post})"
        else:
            boss_of_boss = self.boss.find_boss(boss_name)
            if (boss_of_boss != None):
                return f"{self.name} ({self.post}) подчиняется " +
boss_of_boss
            else:
                return None

    def find_staff (self, staff_name):
        for obj in self.subs:
            if (obj.name == staff_name):
                return f"{self.name} ({self.post}) управляет {obj.name}
({obj.post})"
            else:
                find_staff = obj.find_staff(staff_name)
                if (find_staff != None):
                    return f"{self.name} ({self.post}) управляет " +
find_staff
        return None

class WorkerHosp(Staff):
    def __init__(self, name: str, post: str, boss: AdmHosp):
        self.name = name
        self.post = post
        self.boss = boss
        boss.subs.append(self)

    def print_boss(self):
        print(f"{self.name} подчинён
```

Листинг А.3 — Продолжение листинга А.2

```
{self.boss.name}({self.boss.post})")

    def find_boss(self, boss_name):
        if (self.boss.name == boss_name):
            return f"{self.name} ({self.post}) подчиняется
{self.boss.name} ({self.boss.post})"
        else:
            boss_of_boss = self.boss.find_boss(boss_name)
            if (boss_of_boss != None):
                return f"{self.name} ({self.post}) подчиняется " +
boss_of_boss
            else:
                return None

    def find_staff (self, staff_name):
        return None

class RoomHosp(PsyHospital):
    def __init__(self, name: str, boss: AdmHosp):
        self.name = name
        self.boss = boss
        self.post = "Помещение"
        boss.subs.append(self)

    def print_boss(self):
        print(f"{self.name} управляется
{self.boss.name}({self.boss.post})")

    def find_boss(self, boss_name):
        if (self.boss.name == boss_name):
            return f"{self.name} управляется {self.boss.name}
({self.boss.post})"
        else:
            boss_of_boss = self.boss.find_boss(boss_name)
            if (boss_of_boss != None):
                return f"{self.name} управляется " + boss_of_boss
            else:
                return None

    def find_staff (self, staff_name):
        return None

if __name__ == '__main__':
    # Глава руководства
    Head =          HeadHosp("Иванов И.И.",     "Главный врач")
    # Руководство
    MainNurse =     AdmHosp("Долгова Д.Д.",     "Главная медсестра",
Head)
    HeadOfRooms =   AdmHosp("Дубинкин Д.Д.",    "Заведующий
отделениями",   Head)
    HeadOfStaff =   AdmHosp("Рощин Р.Р.",       "Заведующий
персоналом",    Head)
```

Листинг А.4 — Продолжение листинга А.3

```
    # Штат
    Secur =         WorkerHosp("Листьев Л.Л.",  "Охранник",
HeadOfStaff)
    MedBrat =       WorkerHosp("Петров П.П.",   "Санитар",
MainNurse)
    Cleaner =       WorkerHosp("Сидоров С.С.",  "Уборщик",
HeadOfStaff)
    Nurse =         WorkerHosp("Смирнова С.С.", "Медсестра",
MainNurse)

    # Помещения
    MainHall =      RoomHosp("Главный зал",         HeadOfRooms)
    MedRoom =       RoomHosp("Лечебная палата",     MainNurse)
    Reception =     RoomHosp("Приёмный зал",        HeadOfRooms)
    WalkingArea =   RoomHosp("Прогулочная зона",    HeadOfRooms)

    Head.print_subs()
    MainNurse.print_boss()
    WalkingArea.print_boss()

    print(WalkingArea.find_boss("Иванов И.И."))
    print(Head.find_staff("Приёмный зал"))
```

Приложение Б

Код программы Онтологии

Листинг Б.1 — Основной алгоритм программы

```
import numpy as np

def f(x, y):
    return (1.5 - x + x*y)**2 + (2.25 - x + x*(y**2))**2 + (2.625 - x +
x*(y**3))**2

def otzhig(T = 1,Tmin = 1e-100, alpha = np.random.uniform(0.1, 1)):
    xb = np.random.uniform(-4.5, 4.5)
    yb = np.random.uniform(-4.5, 4.5)

    while (T > Tmin):
        xi = np.random.uniform(-4.5, 4.5)
        yi = np.random.uniform(-4.5, 4.5)

        funci = f(xi,yi)
        funcb = f(xb,yb)

        if (funci - funcb <= 0):
            xb = xi
            yb = yi
        else:
            if (np.exp(-(funci - funcb) / T) > np.random.uniform(0,
1)):
                xb = xi
                yb = yi

        T *= alpha

    return xb, yb
def otzhigKoshi(T = 1,Tmin = 1e-1000, k=1):
    xb = np.random.uniform(-4.5, 4.5)
    yb = np.random.uniform(-4.5, 4.5)

    while (T > Tmin):
        xi = np.random.uniform(-4.5, 4.5)
        yi = np.random.uniform(-4.5, 4.5)

        funci = f(xi,yi)
        funcb = f(xb,yb)

        if (funci - funcb <= 0):
            xb = xi
            yb = yi
```

Листинг Б.2 — Продолжение листинга Б.1

```
else:
            if (np.exp(-(funci - funcb) / T) > np.random.uniform(0,
1)):
                xb = xi
                yb = yi

        T /= k
        k+=1

    return xb, yb

print("Эталон: f(3,0.5) = 0")
xb, yb = otzhig()
print("Отжиг")
print(xb, yb)
print(f(xb, yb))

print("Отжиг Коши")
xb, yb = otzhigKoshi()
print(xb, yb)
print(f(xb, yb))
```

Листинг Б.1.1 — Решение задачи Коммивояжера

```
import numpy as np

def createGraph(cNodes):
    structure = {}
    step = 1
    for i in range (cNodes):
        structure[str(i + step)] = []

    for i in range (cNodes):
        for j in range (i + 1,cNodes):
            w = np.random.randint(1, 6)

            structure[str(i + step)].append((str(j + step), w))
            structure[str(j + step)].append((str(i + step), w))

    return structure, step

def findWay(cNodes, struct, step, T = 1,Tmin = 1e-1000, k = 1):
    start = str(step)
    curr = start
    next_step = ''

    pb = [start]
    wb = 0

    for i in range (cNodes - 1):
        while True:
            next_step = str(np.random.randint(step, cNodes + 1))
            if next_step not in pb:
                break
        pb.append(next_step)

        for l in struct[curr]:
            if l[0] == next_step:
                wb += l[1]

        curr = next_step
    pb.append(start)
    for l in struct[curr]:
            if l[0] == start:
                wb += l[1]

    print(' -> '.join(pb))
    print('Вес:', wb)

    while T > Tmin:
        pi = [start]
        wi = 0
```

Листинг Б.1.2 — Продолжение листинга Б.1.1

```
        for i in range (cNodes - 1):
            while True:
                next_step = str(np.random.randint(step, cNodes + 1))
                if next_step not in pi:
                    break
            pi.append(next_step)

            for l in struct[curr]:
                if l[0] == next_step:
                    wi += l[1]

            curr = next_step
        pi.append(start)
        for l in struct[curr]:
                if l[0] == start:
                    wi += l[1]

        if (wi - wb <= 0):
            wb = wi
            pb = pi
        else:
            if (np.exp(-(wi - wb) / T) > np.random.uniform(0, 1)):
                wb = wi
                pb = pi

        print(' -> '.join(pi))
        print('Вес:', wi)
        T /= k
        k += 1
    return pb, wb


cNodes = 6
structure, step = createGraph(cNodes)
# print(structure)

path, weight = findWay(cNodes, structure, step)
print('\nКратчайший путь:')
print(' -> '.join(path))
print('Вес:', weight)
```

Приложение В

Код программы Онтологии

Листинг В.1 — Основной алгоритм программы

```
import numpy as np

def f(x, y):
    return (1.5 - x + x*y)**2 + (2.25 - x + x*(y**2))**2 + (2.625 - x +
x*(y**3))**2

class Particle():
    def __init__(self):
        self.xi = []
        self.yi = []
        self.func = []
        self.Vxi = []
        self.Vyi = []

    def find_best(self):
        minf = min(self.func)
        for i in range(len(self.func)):
            if minf == self.func[i]:
                return self.xi[i], self.yi[i]


class Roi():
    def __init__(self):
        self.parts = []
        self.glob_best = ()
        self.iteration = 0

        self.c1, self.c2 = 2, 2
        self.r1, self.r2 = np.random.uniform(0, 1),
np.random.uniform(0, 1)

    def create(self, count, minZ, maxZ):
        for _ in range (count):
            part = Particle()

            part.xi.append(np.random.uniform(minZ, maxZ))
            part.yi.append(np.random.uniform(minZ, maxZ))
            part.func.append(f(part.xi[0],part.yi[0]))

            part.Vxi.append(np.random.uniform(minZ, maxZ))
            part.Vyi.append(np.random.uniform(minZ, maxZ))

            self.parts.append(part)
        minf = min(obj.func for obj in self.parts)
        for obj in self.parts:
            self.glob_best = (obj.xi[0], obj.yi[0], obj.func[0]) if
minf == obj.func else self.glob_best
```

Листинг В.2 — Продолжение листинга В.1

```
    def new_v(self, n, xb, yb):
        return (
            self.parts[n].Vxi[self.iteration]
            + self.c1 * self.r1 * (xb -
self.parts[n].xi[self.iteration])
            + self.c2 * self.r2 * (self.glob_best[0] -
self.parts[n].xi[self.iteration])
        ),(
            self.parts[n].Vyi[self.iteration]
            + self.c1 * self.r1 * (yb -
self.parts[n].yi[self.iteration])
            + self.c2 * self.r2 * (self.glob_best[1] -
self.parts[n].yi[self.iteration])
        )

    def make_iter(self, N):
        for i in range(N):
            xb, yb = self.parts[i].find_best()
            Vx, Vy = self.new_v(i, xb, yb)

            self.parts[i].Vxi.append(Vx)
            self.parts[i].Vyi.append(Vy)

            self.parts[i].xi.append(self.parts[i].xi[self.iteration] +
Vx)
            self.parts[i].yi.append(self.parts[i].yi[self.iteration] +
Vy)

self.parts[i].func.append(f(self.parts[i].xi[self.iteration] + Vx,
self.parts[i].yi[self.iteration] + Vy))

        minf = min(min(obj.func) for obj in self.parts)
        for obj in self.parts:
            for i in range (len(obj.func)):
                self.glob_best = (obj.xi[i], obj.yi[i], obj.func[i]) if
minf == obj.func[i] else self.glob_best

        self.iteration += 1

    def print(self):
        for obj in self.parts:
            print(f'x = {obj.xi[self.iteration]}  y =
{obj.yi[self.iteration]}  f(x,y) = {obj.func[self.iteration]}')
```

Листинг В.3— Продолжение листинга В.2

```
if __name__ == '__main__':
    obj = Roi()
    N = 5
    max_iter = 100

    obj.create(N, -4.5, 4.5)

    print("==Начальные позиции роя==")
    obj.print()
    print(obj.glob_best)

    while obj.iteration != max_iter:
        obj.make_iter(N)

    print("\n==Конечные позиции роя==")
    obj.print()
    print(obj.glob_best)
```

Приложение Г

Код программы Муравьиный алгоритм

Листинг Г.1 — Основной алгоритм программы

```
import numpy as np
import random

def createGraph(cNodes):
    structure = {}
    step = 1
    for i in range (cNodes):
        structure[str(i + step)] = []

    for i in range (cNodes):
        for j in range (i + 1,cNodes):
            w = np.random.randint(1, 6)

            structure[str(i + step)].append((str(j + step), w))
            structure[str(j + step)].append((str(i + step), w))

    return structure, step

def aco_tsp(graph, startNode, num_ants=10, iterations=100, alpha=1):
    nodes = list(graph.keys())
    n = len(nodes)

    tau = {i: {j: np.random.uniform(0.1, 1.0) for j, _ in graph[i]} for
i in graph}
    eta = {i: {j: 1 / w for j, w in graph[i]} for i in graph}

    best_length = float('inf')
    best_path = None

    for t in range(iterations):
        paths = []
        lengths = []

        for _ in range(num_ants):
            start = str(startNode)
            unvisited = set(nodes)
            unvisited.remove(start)
            path = [start]
            total_dist = 0
            current = start

            while unvisited:
                neighbors = [n for n, _ in graph[current] if n in
unvisited]
                if not neighbors:
                    break
```

Листинг Г.2 — Продолжение листинга Г.1

```
                probs = []
                denom = sum((tau[current][j] ** alpha) for j in
neighbors)
                for j in neighbors:
                    p = (tau[current][j] ** alpha) / denom
                    probs.append(p)

                next_node = random.choices(neighbors, weights=probs,
k=1)[0]
                dist = next(w for wj, w in graph[current] if wj ==
next_node)
                total_dist += dist
                path.append(next_node)
                unvisited.remove(next_node)
                current = next_node


            if len(path) == n:
                last, first = path[-1], path[0]
                dist = next(w for wj, w in graph[last] if wj == first)
                total_dist += dist
                path.append(first)

            # print(' -> '.join(path))
            # print(total_dist)

            paths.append(path)
            lengths.append(total_dist)

            if total_dist < best_length:
                best_length = total_dist
                best_path = path


        for k in range(num_ants):
            path = paths[k]
            length = lengths[k]
            for i in range(len(path) - 1):
                a, b = path[i], path[i + 1]
                tau[a][b] += 1 / length
                tau[b][a] += 1 / length

    return best_path, best_length
```

Листинг Г.3— Продолжение листинга Г.2

```
if __name__ == '__main__':
    cNodes = 6
    structure, start = createGraph(cNodes)

    path, weight = aco_tsp(structure, startNode=start, num_ants =
cNodes)
    print("Кратчайший путь:", ' -> '.join(path))
    print("Длина:", weight)
```

Приложение Д

Код программы Пчелиного алгоритм

Листинг Д.1 — Основной алгоритм программы

```
import numpy as np

def f(x, y):
    return (1.5 - x + x*y)**2 + (2.25 - x + x*(y**2))**2 + (2.625 - x +
x*(y**3))**2

class Bee():
    def __init__(self):
        self.x = 0
        self.y = 0
        self.f = 0

    def calc(self):
        self.f = f(self.x, self.y)
        return self.f

    def calc_x_area(self, inaccuracy):
        return self.x - inaccuracy, self.x + inaccuracy

    def calc_y_area(self, inaccuracy):
        return self.y - inaccuracy, self.y + inaccuracy
                    break

class Colony():
    def __init__(self):
        self.scouts = 10
        self.best = 5
        self.worst = 2
        self.best_area = 2
        self.worst_area = 1
        self.inaccuracy = 1.5

    def bee_atack(self, iter, minX, maxX, minY, maxY):
        beez = []
        for i in range (iter):
            bee = Bee()
            bee.x = np.random.uniform(minX, maxX)
            bee.y = np.random.uniform(minY, maxY)
            bee.x = np.clip(bee.x, -4.5, 4.5)
            bee.y = np.clip(bee.y, -4.5, 4.5)
            bee.calc()
            beez.append(bee)
        beez.sort(key=lambda b: b.f)
        return beez
```

Листинг Д.2 — Продолжение листинга Д.1

```
    def find_best(self, iter):
        beez = self.bee_atack(self.scouts, -4.5, 4.5, -4.5, 4.5)
        beez = beez[:self.best_area + self.worst_area]

        cur_beez = []
        for i in range(iter):
            for bee in range (self.best_area):
                minX, maxX = beez[bee].calc_x_area(self.inaccuracy)
                minY, maxY = beez[bee].calc_y_area(self.inaccuracy)
                cur_beez += self.bee_atack(self.best, minX, maxX, minY,
maxY)

            for bee in range (self.worst_area):
                minX, maxX = beez[self.best_area +
bee].calc_x_area(self.inaccuracy)
                minY, maxY = beez[self.best_area +
bee].calc_y_area(self.inaccuracy)
                cur_beez += self.bee_atack(self.worst, minX, maxX,
minY, maxY)

            cur_beez.sort(key=lambda b: b.f)
            cur_beez = cur_beez[:self.best_area + self.worst_area]

        return cur_beez[0]


if __name__ == '__main__':
    print("Эталон: f(3, 0.5) = 0")

    c = Colony()
    iter = 1000

    best_bee = c.find_best(iter)
    print(f"Результат x = {best_bee.x:.5f} y = {best_bee.y:.5f} f =
{best_bee.f:.5f}")
```

Приложение Е

Код программы Алгоритм обезьян

Листинг Е.1 — Основной алгоритм программы

```
import numpy as np

def f(x, y):
    return (1.5 - x + x*y)**2 + (2.25 - x + x*(y**2))**2 + (2.625 - x +
x*(y**3))**2

class Monkey:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.f = 0
        self.xi = 0
        self.yi = 0
        self.fi = 0
        self.step = 0.1

    def calc(self):
        self.f = f(self.x, self.y)
        return self.f

    def make_local_jump(self, max_jump):
        self.xi, self.yi,  self.fi = self.x, self.y, self.f
        best_xi, best_yi, best_fi = self.xi, self.yi, self.fi
        jumps_left = max_jump
        while jumps_left > 0:
            trial_x = np.random.uniform(self.xi - self.step, self.xi +
self.step)
            trial_y = np.random.uniform(self.yi - self.step, self.yi +
self.step)
            trial_x = np.clip(trial_x, -4.5, 4.5)
            trial_y = np.clip(trial_y, -4.5, 4.5)
            trial_f = f(trial_x, trial_y)
            if trial_f < best_fi:
                best_xi, best_yi, best_fi = trial_x, trial_y, trial_f
                jumps_left = max_jump
            else:
                jumps_left -= 1

        self.xi, self.yi, self.fi = best_xi, best_yi, best_fi
        return self.fi
```

Листинг Е.2 — Продолжение листинга Е.1

```
    def make_global_jump(self, c_x, c_y):
        self.x = self.x + np.random.uniform(1, 2) * (c_x - self.x)
        self.y = self.y + np.random.uniform(1, 2) * (c_y - self.y)

        self.x = np.clip(self.x, -4.5, 4.5)
        self.y = np.clip(self.y, -4.5, 4.5)

        self.calc()

    def update(self):
        self.x, self.y, self.f = self.xi, self.yi, self.fi


class Troop:
    def __init__(self):
        self.n_monkeys = 15
        self.max_local_jumps = 20

    def monkey_init(self, minZ, maxZ):
        monkeyz = []
        for _ in range(self.n_monkeys):
            m = Monkey()
            m.x = np.random.uniform(minZ, maxZ)
            m.y = np.random.uniform(minZ, maxZ)
            m.calc()

            monkeyz.append(m)
        return monkeyz

    def find_best(self, iter):
        monkeyz = self.monkey_init(-4.5, 4.5)

        for i in range(iter):
            c_x = np.mean([m.x for m in monkeyz])
            c_y = np.mean([m.y for m in monkeyz])

            for monk in monkeyz:
                if monk.make_local_jump(self.max_local_jumps) < monk.f:
                    monk.update()
                else:
                    monk.make_global_jump(c_x, c_y)
                    c_x = np.mean([m.x for m in monkeyz])
                    c_y = np.mean([m.y for m in monkeyz])

        monkeyz.sort(key=lambda b: b.f)
        return monkeyz[0]
```

Листинг Е.3 — Продолжение листинга Е.2

```
if __name__ == '__main__':
    print("Эталон: f(3, 0.5) = 0")

    troop = Troop()
    iter = 100

    best_monkey = troop.find_best(iter)
    print(f"Результат x = {best_monkey.x:.5f} y = {best_monkey.y:.5f} f
= {best_monkey.f:.8f}")
```

Приложение Ж

Код Генетического алгоритма

Листинг Ж.1 — Основной алгоритм программы

```
import numpy as np

def f(x, y):
    return (1.5 - x + x*y)**2 + (2.25 - x + x*(y**2))**2 + (2.625 - x +
x*(y**3))**2


def genetic_minimize_beale(
        pop_size=50,
        n_generations=200,
        x_bounds=(-4.5, 4.5),
        y_bounds=(-4.5, 4.5),
        pc=0.8,
        pm=0.1,
        mutation_sigma=0.1,
        tournament_size=3,
        seed=123
    ):
    rng = np.random.default_rng(seed)

    pop = np.empty((pop_size, 2))
    pop[:, 0] = rng.uniform(x_bounds[0], x_bounds[1], size=pop_size)  #
x
    pop[:, 1] = rng.uniform(y_bounds[0], y_bounds[1], size=pop_size)  #
y

    def fitness(ind):
        x, y = ind
        value = f(x, y)
        return 1.0 / (1.0 + value)
    def evaluate_population(pop):
        return np.array([fitness(ind) for ind in pop])
    def tournament_select(pop, fit):
        idx = rng.integers(0, len(pop), size=tournament_size)
        best_i = idx[np.argmax(fit[idx])]
        return pop[best_i].copy()
    def crossover(parent1, parent2):
        if rng.random() < pc:
            alpha = rng.random()
            child1 = alpha * parent1 + (1 - alpha) * parent2
            child2 = alpha * parent2 + (1 - alpha) * parent1
        else:
            child1, child2 = parent1.copy(), parent2.copy()
        return child1, child2
```

Листинг Ж.2 — Продолжение листинга Ж.1

```
        def mutate(ind):
            for i in range(len(ind)):
                if rng.random() < pm:
                    ind[i] += rng.normal(0.0, mutation_sigma)
            ind[0] = np.clip(ind[0], x_bounds[0], x_bounds[1])
            ind[1] = np.clip(ind[1], y_bounds[0], y_bounds[1])
            return ind
        best_ind = None
        best_fit = -np.inf
        for gen in range(n_generations):
            fit = evaluate_population(pop)

            gen_best_i = np.argmax(fit)
            if fit[gen_best_i] > best_fit:
                best_fit = fit[gen_best_i]
                best_ind = pop[gen_best_i].copy()

            print(f"f = {f(best_ind[0], best_ind[1]):.6f} в точке
{best_ind}")

            new_pop = []

            while len(new_pop) < pop_size:
                p1 = tournament_select(pop, fit)
                p2 = tournament_select(pop, fit)

                c1, c2 = crossover(p1, p2)

                c1 = mutate(c1)
                c2 = mutate(c2)

                new_pop.append(c1)
                if len(new_pop) < pop_size:
                    new_pop.append(c2)

            pop = np.array(new_pop)

        best_x, best_y = best_ind
        best_value = f(best_x, best_y)
        return best_x, best_y, best_value


if __name__ == "__main__":
    x_opt, y_opt, f_opt = genetic_minimize_beale(
        pop_size=80,
        n_generations=300
    )
    print(f"Результат x = {x_opt:.5f} y = {y_opt:.5f} f = {f_opt:.8f}")
```